Using finite transducers for describing and synthesising structural time-series constraints

Nicolas Beldiceanu, Mats Carlsson, Rémi Douence, Helmut Simonis

Constraints manuscript No.
(will be inserted by the editor)

Using Finite Transducers for Describing and Synthesising Structural Time-Series Constraints

Nicolas Beldiceanu · Mats Carlsson · Rémi Douence · Helmut Simonis

July 30, 2015

Abstract We describe a large family of constraints for structural time series by means of function composition. These constraints are on aggregations of features of patterns that occur in a time series, such as the number of its peaks, or the range of its steepest ascent. The patterns and features are usually linked to physical properties of the time series generator, which are important to capture in a constraint model of the system, i.e. a conjunction of constraints that produces similar time series. We formalise the patterns using finite transducers, whose output alphabet corresponds to semantic values that precisely describe the steps for identifying the occurrences of a pattern. Based on that description, we automatically synthesise automata with accumulators, as well as constraint checkers. The description scheme not only unifies the structure of the existing 30 time-series constraints in the Global Constraint Catalogue, but also leads to over 600 new constraints, with more than 100,000 lines of synthesised code.

Keywords global constraint · time series · Global Constraint Catalogue · constraint synthesis · finite transducer

1 Introduction

A *time series* is here a sequence of integers, corresponding to measurements taken over a time interval. Time series are ubiquitous in many application areas. In our current

Nicolas Beldiceanu
TASC (CNRS/INRIA), Mines Nantes, FR – 44307 Nantes, France
E-mail: Nicolas.Beldiceanu@mines-nantes.fr

Mats Carlsson
SICS, P.O. Box 1263, SE – 164 29 Kista, Sweden
E-mail: Mats.Carlsson@sics.se

Rémi Douence
ASCOLA (CNRS/INRIA), Mines Nantes, FR – 44307 Nantes, France
E-mail: Remi.Douence@mines-nantes.fr

Helmut Simonis
Insight Centre for Data Analytics, University College Cork, Ireland
E-mail: Helmut.Simonis@insight-centre.org

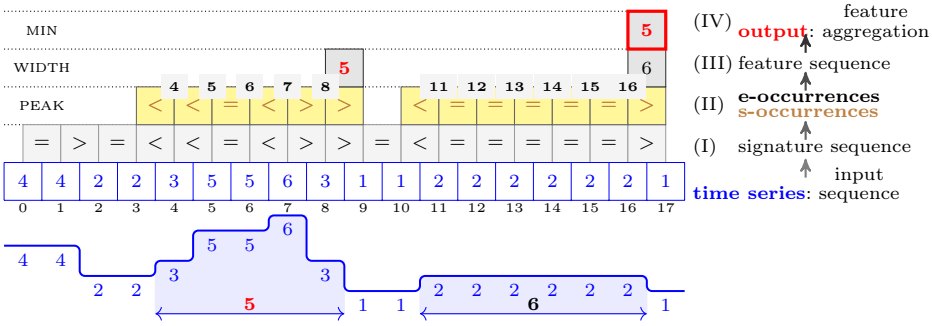


Fig. 1 Describing time-series constraints as a function composition, exemplified on the MIN_WIDTH_PEAK$(5, \langle 4,4,2,2,3,5,5,6,3,1,1,2,2,2,2,2,1\rangle)$ constraint instance of Example 1 in Section 2.1: (I) building the signature sequence by comparing adjacent input values; (II) finding all occurrences of the `peak` pattern, i.e., maximal words matching the regular expression $<\ (=\mid <)^*\ (>\mid =)^*\ >$, and computing the corresponding e-occurrences; (III) computing the `width` feature of each peak from the e-occurrences; and (IV) aggregating the feature values using the `Min` aggregator.

work for example, we use them to model the power output of electric power stations over multiple days, to describe environmental data (temperature, humidity, CO_2 level) in building management, or to model the capacity of a hospital clinic for each day over a period of years. These type of time series are constrained by physical or organizational (in general *structural*) limits, which restrict the evolution of the series. We use the term structural in the sense of [18], which defines "A structural time series model is one which is set up in terms of components which have a direct interpretation.". In our view these components are constraints, which we express as a combination of *patterns*, *features* and *aggregators*.

We have identified a set of, at the moment, 20 *patterns* that capture important structural information about the time series. For example, a *peak* is a maximal subsequence of non-strict increases followed by non-strict decreases, delimited to the left and right by a strict increase and a strict decrease, respectively.

A *feature* is obtained by applying a total function to a pattern occurrence. For example, the *width* is the number of elements of the pattern occurrence.

The *aggregation* of the features of all the occurrences of the same pattern in a time series is obtained by reducing the feature sequence through another total function, such as the minimum. For example, the time series given by the integer values $4_0, 4_1, 2_2, 2_3, 3_4, 5_5, 5_6, 6_7, 3_8, 1_9, 1_{10}, 2_{11}, 2_{12}, 2_{13}, 2_{14}, 2_{15}, 2_{16}, 1_{17}$, where the indices give the positions within the sequence, contains two peaks, namely $3_4, 5_5, 5_6, 6_7, 3_8$ and $2_{11}, 2_{12}, 2_{13}, 2_{14}, 2_{15}, 2_{16}$, of widths 5 and 6 respectively, so that the narrowest peak has width 5, as illustrated in Figure 1.

Research on time series has a long tradition. Classical research focuses on extracting meaningful statistics towards predicting future values or mining the time series [13, 23]. More recently, the keen interest of data science for a better understanding of user behaviour gave renewed relevance to building models from time series [17,21]. From a constraint programming perspective, work on time series was initiated by [16] in the context of mining. More recently, some 30 time-series constraints were introduced in the Global Constraint Catalogue [3,6] in order to use them as a vocabulary, or *bias*, for learning constraint models from electricity production curves [8].

The **contribution** of this paper is twofold. First, we provide a systematic way to describe structural time-series constraints using one main ingredient, which we call a *seed transducer*. Informally, a seed transducer is a finite-state transducer [11,24], that is a finite-state automaton that produces an output sequence from its input sequence, where the output alphabet consists of letters that describe the phases of finding a pattern. Second, we show how to use a seed transducer to synthesise new time-series constraints automatically for the Global Constraint Catalogue: we synthesise a checker and a very small automaton with accumulators [2], typically with at most 5 states, for each constraint, as well as the meta data used in the catalogue, so that our synthesised time-series constraints can be directly used by the Constraint Seeker [9] and Model Seeker [10]. Synthesising fast checkers goes back to the work on Rabbit [19], and there is recent work on synthesising propagators for the TABLE constraint [15] as well as CSP-solvers [25].

In Section 2, we show how to describe a variety of structural time-series constraints via a description made of four layers. In Section 3, we then introduce the notion of seed transducer used for describing how all occurrences of a pattern are found for a time series. For the two classes of time-series constraints presented in Section 4, we explain in Section 5 how to synthesise automatically an automaton with accumulators [2]. In Section 6, we provide use cases in the context of learning constraint models on time series. Finally, we conclude in Section 7.

2 A Four-Layered Description of Time-Series Constraints

Our focus is on time-series constraints that are defined as total-function constraints [4] on a sequence of variables. We first give the intuition of how to describe concisely a class of such structural time-series constraints by a four-layered scheme. We then formally define the notion of pattern, which is a key ingredient for describing time-series constraints.

2.1 Intuition: Signature, Pattern, Feature, and Aggregation

Given a pattern and a time series $x_0, x_1, \ldots, x_{n-1}$ of integer constants, called the *input values* and forming the *input sequence*, a single integer is computed in four consecutive steps, which we now describe:

I. Compare each pair of adjacent input values in order to build a sequence $s_0, s_1, \ldots, s_{n-2}$ of *signature values* over the alphabet $\{<, =, >\}$, as follows: $(x_i < x_{i+1} \Leftrightarrow s_i = \text{`}<\text{'}) \wedge (x_i = x_{i+1} \Leftrightarrow s_i = \text{`}=\text{'}) \wedge (x_i > x_{i+1} \Leftrightarrow s_i = \text{`}>\text{'})$. The signature values form the *signature sequence*.

II. Within the signature sequence, find *all* maximal words [1], or *s-occurrences*, matching a regular expression corresponding to the pattern of interest. For example, a peak (as defined in Section 1) matches '$<\ (=\ |\ <)^*(>\ |\ =)^*\ >$'. In Section 2.2, we will call *e-occurrence* the input index subsequence that yields an s-occurrence. For example, the peak $<_0, <_1, =_2, <_3, >_4, >_5$ for the input sequence $2_0, 3_1, 5_2, 5_3, 6_4, 3_5, 1_6$ has $[[1..5]]$, short for $1, 2, 3, 4, 5$, as e-occurrence.

III. For each found pattern occurrence, compute an integer *feature value*, so that we obtain a *feature sequence*. The features we currently consider are `one`, `width`, `surface`,

pattern	regular expression r	before b	after a
`increasing`	$<$	0	0
`increasing_sequence`	$<(<\mid=)^*<\mid<$	0	0
`increasing_terrace`	$<=^+<$	1	1
`summit`	$(<\mid(<(=\mid<)^*<))(>\mid(>(=\mid>)^*>))$	1	1
`plateau`	$<=^*>$	1	1
`proper_plateau`	$<=^+>$	1	1
`strictly_increasing_sequence`	$<^+$	0	0
`peak`	$<(=\mid<)^*(>\mid=)^*>$	1	1
`inflexion`	$<(<\mid=)^*>\mid>(>\mid=)^*<$	1	1
`steady`	$=$	0	0
`steady_sequence`	$=^+$	0	0
`zigzag`	$(<>)^+(<\mid<>)\mid(><)^+(>\mid><)$	1	1

Table 1 Pattern list; by permuting the symbols '<' and '>' in the regular expressions, we get `decreasing`, `decreasing_sequence`, `decreasing_terrace`, `gorge`, `plain`, `proper_plain`, `strictly_decreasing_sequence`, and `valley` as counterparts of the first eight patterns, so that there are twenty patterns in total.

`min`, `max`, and `range`, and correspond for the given e-occurrence, denoted e, to, respectively, the value 1, to the number $|e|$ of elements of e, to $\sum_{i\in e} x_i$, to $\min_{i\in e} x_i$, to $\max_{i\in e} x_i$, and to $\max_{i\in e} x_i - \min_{i\in e} x_i$.

IV. Aggregate the values of the feature sequence into a single integer value. The aggregators we currently consider are summing up (`Sum`), taking the minimum (`Min`), and taking the maximum (`Max`). The feature `one` only makes sense with the `Sum` aggregator.

As a convention we define a name of the constraint which is the concatenation of the aggregator function, the feature name and the pattern. For the combination of aggregator function `Sum`, and the feature `one`, which counts the occurrences of a pattern, we use the notation NB_.

Example 1 We consider the MIN_WIDTH_PEAK constraint from Figure 1, which constrains the minimum width of the peaks of a time series. The given time series reveals two peaks, with e-occurrences [[4..8]] and [[11..16]] of widths 5 and 6 respectively, so that the minimum width is 5.

2.2 Patterns and Occurrences

Patterns describe the *topological* aspect of subsequences of a time series, as only adjacent values of the time series are compared.

Definition 1 (Pattern) A *pattern* p over the alphabet $\{<,=,>\}$ is a triple $\langle r, b, a\rangle$, where r is a regular expression over $\{<,=,>\}$ that is only matched by non-empty words, while b and a are two non-negative integers, whose role will be explained in Definition 2.

In Definition 1, b and a are intended to delete parts of the pattern that are used to detect the start and end of a pattern, but which should not be part of the feature computation. The 20 patterns we consider are provided in Table 1, with illustrations in Figure 2.

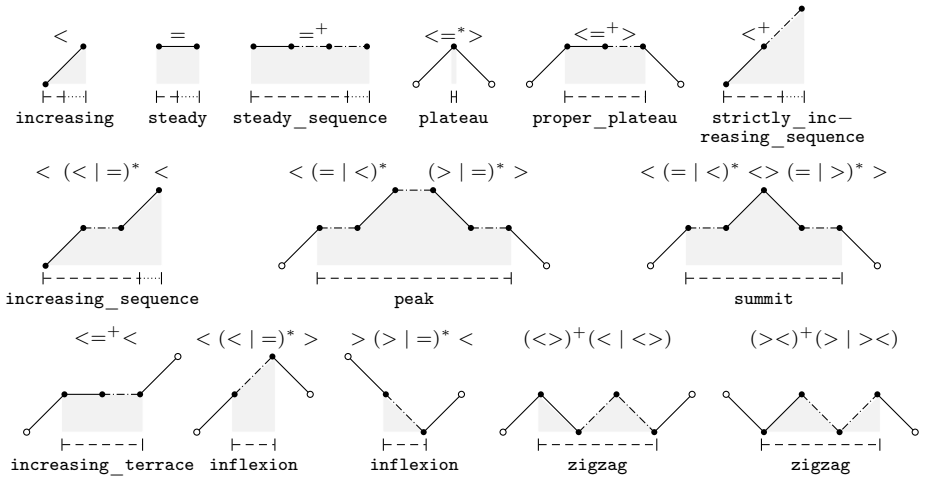


Fig. 2 Illustration of the patterns of Table 1, with time on the horizontal axis and the measurements on the vertical axis: only the relative vertical positions of adjacent points matter, not their magnitudes. An i-occurrence is shown with a dashed line; its extension, when it exists, to the e-occurrence is shown with a dotted line: filled points are part of the e-occurrence, but not the hollow points. Dash-dotted lines include an arbitrary number of points. Grey-shaded areas approximate the pattern surface.

Definition 2 (s-occurrence, i-occurrence, e-occurrence) Given an input sequence $x_0, x_1, \ldots, x_{n-1}$, its signature sequence $S = s_0, s_1, \ldots, s_{n-2}$, a pattern $\langle r, b, a \rangle$, and a non-empty signature subsequence $s_i, s_{i+1}, \ldots, s_j$, with $0 \leq i \leq j \leq n-2$, forming a maximal word that matches r, the *s-occurrence* $(i..j)$ is the index sequence $i, i+1, \ldots, j$; the *i-occurrence* $[(i+b)..j]$ is the index sequence $i+b, \ldots, j$; and the *e-occurrence* $[[(i+b)..(j+1-a)]]$ is the index sequence $i+b, \ldots, j+1-a$.

Thus, an s-occurrence identifies an occurrence of a pattern in a signature sequence. An i-occurrence identifies an occurrence of a pattern in an input sequence. Note that i-occurrences of the *same* pattern of Table 1 never overlap: e.g., i-occurrences of `increasing` or `steady` never overlap since they consist of a single index, while i-occurrences of `plateau` or `proper_plateau` never overlap since their regular expressions start with a symbol that is not repeated within the regular expression. This property will be used in Section 4.1 to design a constraint that links a time series with the indices of all occurrences of a pattern.

An e-occurrence is used for computing the feature value of a pattern occurrence of Table 1 (as seen in Section 2.1), and may differ from the i-occurrence.

Example 2 Consider the input sequence $1_0, 2_1, 3_2, 3_3, 3_4, 4_5, 4_6$ and its signature sequence $<_0, <_1, =_2, =_3, <_4, =_5$:

- The `increasing_terrace` pattern has '$<=^+<$' as regular expression. There is one terrace, namely $<_1, =_2, =_3, <_4$. The attributes $b = 1 = a$ exclude the first and last input values, namely 2_1 and 4_5, for its i-occurrence and e-occurrence, which thus are [2..4] and [[2..4]] corresponding to the flat part of the pattern as shown in Part (A) of Figure 3.
- The `increasing` pattern has '$<$' as regular expression. There are three increases, namely $<_0$, and $<_1$, and $<_4$ as shown in Part (B). Since the attribute b is equal to

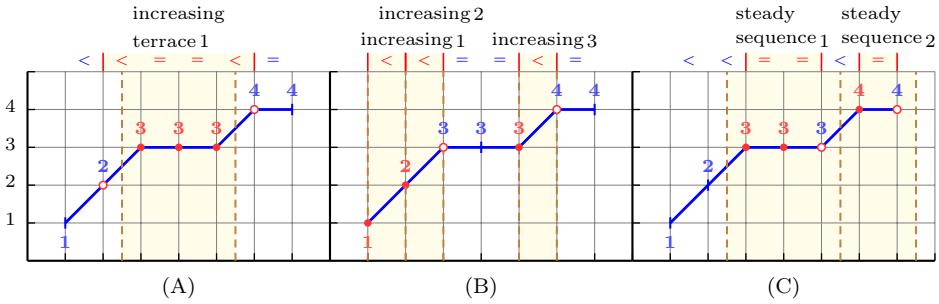


Fig. 3 Illustrating the three patterns of Example 2: s-occurrence are shown on top of each sub-figure, i-occurrence are shown with plain circles, e-occurrences correspond to those plain and hollow circles that are located in the yellow parts between (or crossing) two consecutive dashed lines.

zero, the i-occurrences match the s-occurrences, and are therefore [0..0], [1..1], and [4..4]. To compute the feature values correctly, as the attributes b and a are both zero, both the first and the second input values corresponding to the increases are included for their e-occurrences, which thus are [[0..1]], [[1..2]], and [[4..5]].

– The `steady_sequence` pattern has '$=^+$' as regular expression. There are two steady sequences, namely $=_2, =_3$ and $=_5$ as shown in Part (C). The attributes $b = 0 = a$ include the first and last input values corresponding to the steady sequences for their e-occurrences, which therefore are [[2..4]] and [[5..6]], while their i-occurrences are just [2..3] and [5..5], matching the s-occurrences.

3 Seed Transducers

Recall that a *deterministic finite transducer* [24] is a tuple $\langle Q, \Sigma, \Sigma', \delta, q_0, A\rangle$, where Q is the set of *states*, Σ the *input alphabet*, Σ' the *output alphabet*, $\delta\colon Q \times \Sigma \to Q \times \Sigma'$ the *transition function*, which must be total, $q_0 \in Q$ the *start state*, and $A \subseteq Q$ the set of *accepting states*. When $\delta(q, \sigma) = \langle q', \sigma'\rangle$, there is a transition from state q to state q' upon reading symbol σ in the input of the transducer and writing the symbol σ' to the output of the transducer: we write this as $q \xrightarrow{\sigma:\sigma'} q'$. A *deterministic finite automaton* (DFA) is a transducer without an output alphabet. In a graphical representation of a transducer or automaton, we indicate the start state by an arrow coming from nowhere.

Each pattern of Table 1 is represented by what we call a *seed transducer*, whose aim is to describe the way the i-occurrences are found for an input sequence $x_0, x_1, \ldots, x_{n-1}$. A seed transducer actually reads the corresponding symbolic [26] signature sequence $s_0, s_1, \ldots, s_{n-2}$ and produces a sequence $\tau_0, \tau_1, \ldots, \tau_{n-2}$ of symbols, whose purpose is to guide the synthesis of code for the constraint classes we will give in Section 4. Before defining the notion of seed transducer, we introduce the symbols of its output alphabet, called the *semantic alphabet*:

– $\tau_i =$ **found** means that index i is inside a new i-occurrence, which may have started before i and may continue after i.
– $\tau_i =$ **found**$_{\text{end}}$ means that index i is inside a new i-occurrence, which may have started before i but which ends with i.

– $\tau_i = \mathbf{maybe}_{\text{before}}$ means that index i may belong to an i-occurrence, but that this must be confirmed by producing a '**found**' or '$\mathbf{found}_{\text{end}}$' while reading $s_{i+1}, \ldots, s_{n-2}$.
– $\tau_i = \mathbf{out}_{\text{reset}}$ with $\tau_{i-k} = \tau_{i-k+1} = \cdots = \tau_{i-1} = \mathbf{maybe}_{\text{before}}$ ($i = k \vee \tau_{i-k-1} \neq \mathbf{maybe}_{\text{before}}$) means that index i is outside any i-occurrence and that indices from $i-k$ to $i-1$ are also outside any i-occurrence.
– $\tau_i = \mathbf{in}$ means that index i is inside an i-occurrence for which a '**found**' was already produced.
– $\tau_i = \mathbf{maybe}_{\text{after}}$ means that index i may belong to an i-occurrence for which a '**found**' was already produced, but that this must be confirmed by producing $\mathbf{maybe}^*_{\text{after}}\mathbf{in}$ while reading a prefix of $s_{i+1}, \ldots, s_{n-2}$.
– $\tau_i = \mathbf{out}_{\text{after}}$ means that index i is outside any i-occurrence, but that an i-occurrence has ended at index $i-1$.
– $\tau_i = \mathbf{out}$ means that index i is outside any i-occurrence and that τ_{i-1} is neither a $\mathbf{maybe}_{\text{before}}$ nor a $\mathbf{maybe}_{\text{after}}$.

For conciseness, the subscripts 'after', 'before', 'end', and 'reset' will be abbreviated by their first letters. Examples will be given after the next definition.

Definition 3 (Seed Transducer) A *seed transducer* is a transducer with output alphabet $\{\mathbf{found}, \mathbf{found}_{\text{e}}, \mathbf{in}, \mathbf{maybe}_{\text{b}}, \mathbf{maybe}_{\text{a}}, \mathbf{out}, \mathbf{out}_{\text{a}}, \mathbf{out}_{\text{r}}\}$, input alphabet $\{<, =, >\}$, and only accepting states.

Example 3 See Parts B, C, and D of Figure 4 for seed transducers of the `increasing`, `plateau`, and `peak` patterns of Table 1. The latter works as follows: one starts from state d and stays there until an increase ($<$) transits to state r; one stays in state r until a decrease ($>$) transits to state t and signals that a peak was found; one stays in state t until an increase transits back to state r; for the input sequence $4_0, 4_1, 2_2, 2_3, 3_4, 5_5, 5_6, 6_7, 3_8, 1_9, 1_{10}, 2_{11}, 2_{12}, 2_{13}, 2_{14}, 2_{15}, 2_{16}, 1_{17}$, the transitions are as follows:

$$d \xrightarrow{4=4:\mathbf{out}} d \xrightarrow{4>2:\mathbf{out}} d \xrightarrow{2=2:\mathbf{out}} d \xrightarrow{2<3:\mathbf{out}} r \xrightarrow{3<5:\mathbf{maybe}_{\text{b}}} r \xrightarrow{5=5:\mathbf{maybe}_{\text{b}}} r$$
$$\xrightarrow{5<6:\mathbf{maybe}_{\text{b}}} r \xrightarrow{6>3:\mathbf{found}} t \xrightarrow{3>1:\mathbf{in}} t \xrightarrow{1=1:\mathbf{maybe}_{\text{a}}} t \xrightarrow{1<2:\mathbf{out}_{\text{a}}} r \xrightarrow{2=2:\mathbf{maybe}_{\text{b}}} r$$
$$\xrightarrow{2=2:\mathbf{maybe}_{\text{b}}} r \xrightarrow{2=2:\mathbf{maybe}_{\text{b}}} r \xrightarrow{2=2:\mathbf{maybe}_{\text{b}}} r \xrightarrow{2=2:\mathbf{maybe}_{\text{b}}} r \xrightarrow{2>1:\mathbf{found}} t$$

The two '**found**' correspond to two peaks: the first peak corresponds to the word from the first '$\mathbf{maybe}_{\text{b}}$' to the first '**in**' (i.e., the word '$\mathbf{maybe}^3_{\text{b}}$ **found in**') and its i-occurrence is [4..8]; the second peak corresponds to the word from just after the last '$\mathbf{out}_{\text{a}}$' to the last '**found**' (i.e., the word '$\mathbf{maybe}^5_{\text{b}}$ **found**') and its i-occurrence is [11..16].

Definition 4 (t-occurrence) Given a seed transducer $\mathcal{S}$ and a signature sequence s, the *t-occurrence* of $\mathcal{S}$ for s consists of the indices of the semantic letters of a maximal word within the transduction of s that matches one of the regular expressions '$\mathbf{maybe}^*_{\text{b}}\mathbf{found}_{\text{e}}$' and '$\mathbf{maybe}^*_{\text{b}}\mathbf{found}(\mathbf{maybe}^*_{\text{a}}\mathbf{in}^+)^*$'.

Definition 5 (Seed Transducer Wellformedness) A seed transducer $\mathcal{S}$ is *well-formed* with respect to a pattern *pat* if the following conditions hold:

– All its produced words are accepted by the DFA in Part A of Figure 4.

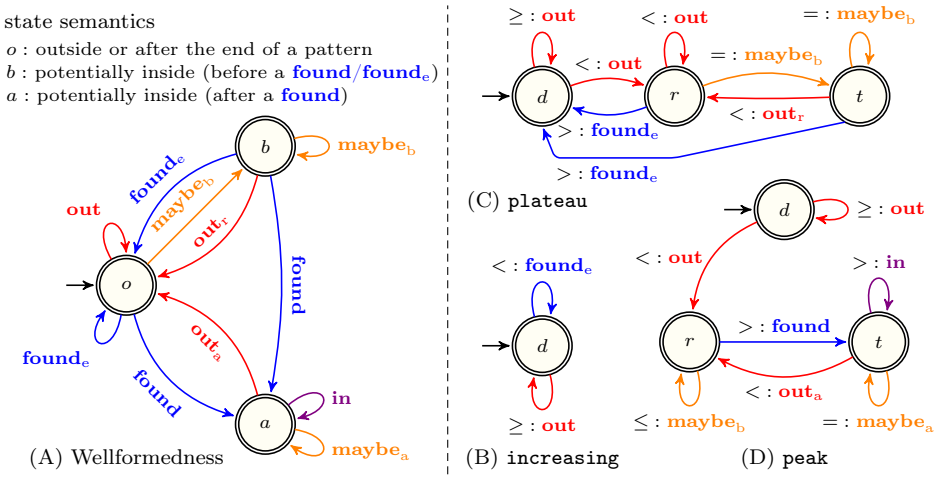


Fig. 4 (A) DFA defining the output language of a well-formed seed transducer. (B,C,D) Seed transducers, involving all eight letters of the semantic alphabet, of the `increasing`, `plateau`, and `peak` patterns, with regular expressions '<', '<=*>', and '< (= | <)*(> | =)* >' respectively; a self-loop labelled by ≥ (respectively ≤) is a shortcut for two self-loops labelled by > and = (respectively < and =).

– The t-occurrence of $\mathcal{S}$ for any input sequence x coincides with the i-occurrence of pattern *pat* for x.

Example 4 The seed transducer in Figure 4D of the `peak` pattern is well-formed since (1) its output language is a subset of the language of the DFA in Figure 4A, and (2) its input language producing '$\mathbf{maybe}_b^*\mathbf{found}(\mathbf{maybe}_a^*\mathbf{in}^+)^*$', namely '(< | =)* > ((> | =)* >$^+$)*', is equivalent to the regular expression associated with the `peak` pattern in Table 1, namely '< (= | <)*(> | =)* >', from which we remove the first letter since `peak` has $b = 1$.

Seed transducers will be used in Section 5 to synthesise automata with accumulators for the constraint classes that the next section introduces.

4 Constraint Classes

We now introduce the two main constraint classes we associate to a pattern. Both are defined as total functions on a sequence. Since we are describing constraints rather than ground instances thereof, we now switch from values to variables, and from functions to total-function constraints.

The first class allows identifying the i-occurrences of a pattern in a sequence (e.g., it allows identifying the i-occurrences of a peak). The second class computes a result from the e-occurrences of pattern in a sequence (e.g., it computes the minimum width of the peaks of a sequence as illustrated in Example 1). Other constraints such as the minimum or maximum distance between consecutive i-occurrences of a pattern or the comparison of feature values of consecutive e-occurrences of a pattern have also been introduced but are outside the scope of this paper due to space limits. Altogether these

constraint classes allow us to cover 28 of the 30 time-series constraints of the Global Constraint Catalogue[1] and to synthesise them in a systematic way leading to more than 600 time-series constraints.

4.1 Footprint Constraint

Footprint constraints allow us to state constraints on the occurrence or non-occurrence of a pattern in a specific time interval. For instance, in the context of energy production, it is quite common to identify time intervals where we know that there will be a production peak balancing a known consumption peak. Footprint constraints also allow reporting precisely time intervals where anomalies like a zigzag pattern occur.

Definition 6 The FOOTPRINT$(pat, \langle x_0, x_1, \ldots, x_{n-1}\rangle, \langle p_0, p_1, \ldots, p_{n-1}\rangle)$ constraint, where pat is the name of one of the patterns of Table 1, $x_0, x_1, \ldots, x_{n-1}$ is a sequence of integer variables, and $p_0, p_1, \ldots, p_{n-1}$ is a sequence of integer variables between 0 and n, holds if:

- $p_k = 0$ if index k does not occur in any i-occurrence of pattern pat in the input sequence $x_0, x_1, \ldots, x_{n-1}$,
- $p_k = j > 0$ if index k belongs to the j^{th} i-occurrence of pattern pat when reading the sequence $x_0, x_1, \ldots, x_{n-1}$.

Example 5 If we consider the time series introduced in Example 1, the corresponding FOOTPRINT$(\texttt{peak}, \langle 4_0, 4_1, 2_2, 2_3, 3_4, 5_5, 5_6, 6_7, 3_8, 1_9, 1_{10}, 2_{11}, 2_{12}, 2_{13}, 2_{14}, 2_{15}, 2_{16}, 1_{17}\rangle$, $\langle 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 2, 2, 2, 2, 2, 2, 0\rangle)$ constraint holds since:

- Within the third argument, the first stretch of five 1s corresponds to the first i-occurrence of peak, namely [4..8].
- The second stretch of six 2s coincides with the second i-occurrence of peak, namely [11..16].

4.2 Constraint on the Aggregation of Pattern Features

Definition 7 The AGGREGATE_FEATURE_PATTERN$(pat, f, g, res, \langle x_0, x_1, \ldots, x_{n-1}\rangle)$ constraint, where pat is one of the pattern names of Table 1, $f \in \{\texttt{one}, \texttt{width}, \texttt{surface}, \texttt{min}, \texttt{max}, \texttt{range}\}$ is a feature, $g \in \{\texttt{Min}, \texttt{Max}, \texttt{Sum}\}$ is an aggregator, res is an integer variable, and $x_0, x_1, \ldots, x_{n-1}$ is a sequence of integer variables, holds if the following two conditions hold:

- If pattern pat does not occur in $x_0, x_1, \ldots, x_{n-1}$, then res is the default value $\text{default}_{g,f}$ associated with the pair $\langle g, f\rangle$ in Table B of Figure 6.
- Otherwise, res is the aggregation with respect to g of the feature values of the occurrences of pattern pat in $x_0, x_1, \ldots, x_{n-1}$.

For a given triple $\langle pat, f, g\rangle$, the AGGREGATE_FEATURE_PATTERN constraint is named by concatenating the aggregation operator name g, the feature name f, and the pattern name pat, like we did in Example 1 with the MIN_WIDTH_PEAK$(res, \langle x_0, x_1, \ldots, x_{n-1}\rangle)$ constraint.

[1] In fact, all existing time-series constraints except BIG_PEAK and BIG_VALLEY [6] fit into our framework. For these two constraints, our seed transducers must be extended with guarded transitions.

5 Synthesising Footprint Constraint and Aggregation Constraint of Pattern Feature

Based on the transducers of the patterns, we synthesise several hundred constraints, which are all represented as automata with accumulators [2], as well as their corresponding checkers as Prolog programs.

The key idea for doing such a synthesis is to use what we call a *decoration table*, which is independent from the seed transducers we consider. The decoration table introduces a set of accumulators with their initial values and defines for each letter of the semantic alphabet how to update these accumulators. Then the synthesis process replaces the semantic letter of each transition of a seed transducer by the accumulator update instructions.

Following [5], we first recall the notion of automata with accumulators. We then provide the decoration tables associated with the FOOTPRINT constraint and the AGGREGATION constraint of pattern feature. For the later case we prove that the generated automaton with accumulators returns the expected result.

5.1 Background: Automata with Accumulators

We here define a *memory-DFA* (mDFA) with a memory of $k \geq 0$ accumulators as a tuple $\langle Q, \Sigma, \delta, q_0, I, A, \alpha \rangle$, where Q, Σ, q_0, and A are as in a DFA, while the transition function δ has signature $(Q \times \mathbb{Z}^k) \times \Sigma \to Q \times \mathbb{Z}^k$, and similarly for its extended version $\widehat{\delta}$. Let Σ^* denote the infinite set of words built from Σ ($\Sigma = \{<, =, >\}$ in our case), including the empty word, denoted ϵ. The *extended transition function* $\widehat{\delta}\colon Q \times \Sigma^* \to Q$ for words (instead of symbols) is recursively defined by $\widehat{\delta}(q, \epsilon) = q$ and $\widehat{\delta}(q, w\sigma) = \delta(\widehat{\delta}(q, w), \sigma)$ for a word w and symbol σ. A word w is *accepted* if $\widehat{\delta}(q_0, w) \in A$. On a transition, there can be a *guard*, that is a comparison between accumulators or variables, which may enable or disable the transition [12, page 452]. Further, I is the k-tuple of initial values of the variables in the memory. Finally, $\alpha\colon A \times \mathbb{Z}^k \to \mathbb{Z}$ is called the *acceptance function* and transforms the memory of an accepting state into an integer. Given a word w, the mDFA returns $\alpha(\widehat{\delta}(\langle q_0, I \rangle, w))$ if w is accepted. Note that δ, $\widehat{\delta}$, and α are total functions.

5.2 Synthesising Footprint Constraints

To get the footprint constraint of a pattern *pat*, i.e., an automaton with accumulators, from its seed transducer $\mathcal{S}_{pat}$ we replace the semantic annotations of each transition of $\mathcal{S}_{pat}$ by using Table A of Figure 5. Table A provides for each type of semantic letter the corresponding replacement, where C is an accumulator used for identifying each occurrence of pattern *pat*:

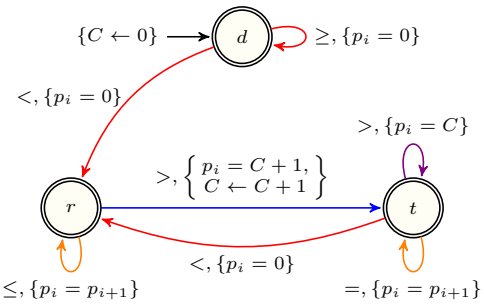
- When the i^{th} semantic letter τ_i is **out**, $\mathbf{out}_{\text{r}}$, or $\mathbf{out}_{\text{a}}$, index i cannot belong to an i-occurrence, and consequently p_i is set to 0.
- When τ_i is $\mathbf{maybe}_{\text{b}}$ (respectively $\mathbf{maybe}_{\text{a}}$), indices i and $i+1$ either belong both to the same i-occurrence or do not both belong to any i-occurrence, depending on whether they are followed or not by a **found** or $\mathbf{found}_{\text{e}}$ (respectively **in**). Consequently p_i and p_{i+1} are made equal.

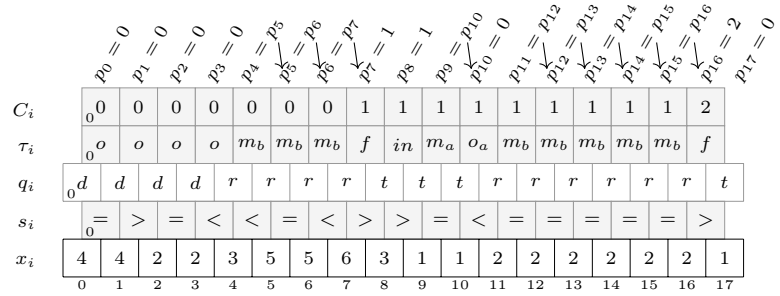
(A)

initialisation	$C \leftarrow 0$
return	$p_n = 0$

semantic letters	**annotations**	
	guard	*update of* C
out	$p_i = 0$	
out$_{\text{r}}$	$p_i = 0$	
out$_{\text{a}}$	$p_i = 0$	
maybe$_{\text{b}}$	$p_i = p_{i+1}$	
maybe$_{\text{a}}$	$p_i = p_{i+1}$	
found$_{\text{e}}$	$p_i = C + 1$	$C \leftarrow C + 1$
found	$p_i = C + 1$	$C \leftarrow C + 1$
in	$p_i = C$	

(B)





(C)

Fig. 5 (A) Decoration table for synthesising the footprint automaton from the semantic letters attached to the transitions of the seed transducer, (B) Footprint automaton of the `peak` pattern obtained by replacing the output semantic letters of the seed transducer of the seed automaton of the `peak` pattern (see Part D of Figure 4) by the corresponding annotations given by the decoration table in (A), (C) Illustrates the execution of the footprint automaton of the `peak` pattern where x_i, s_i, q_i, τ_i, C_i are the sequence, signature, state, semantic, and accumulator variables; the tiny 0 in the left lower corners corresponds to the index of the corresponding variable, and o, m_b, f, m_a, o_a are shortcuts for **out**, **maybe**$_{\text{b}}$, **found**, **maybe**$_{\text{a}}$, **out**$_{\text{a}}$.

– When τ_i is **found** or **found**$_{\text{e}}$, index i belongs to a new i-occurrence. Consequently p_i is set to $C+1$ and we increment the number of i-occurrences already encountered.
– When τ_i is **in**, index i belongs to the C^{th} i-occurrence.

Part (B) of Figure 5 shows the generated constraint for the `peak` pattern.

5.3 Synthesising Aggregation of Pattern Feature Constraints

Similarly to what we did for generating footprint constraints, Table C of Figure 6 gives the decoration table for synthesising aggregation of pattern feature constraints with respect to a feature $f \in \{\texttt{one}, \texttt{width}, \texttt{surface}, \texttt{max}, \texttt{min}\}$ and an aggregation operator $g \in \{\texttt{Max}, \texttt{Min}, \texttt{Sum}\}$ for patterns for which the **after** attribute is set to 1 (see the last column of Table 1).[2] Note that from Definition 2, if the **after** attribute is set to 1,

[2] A very similar decoration table for handling the case when the **after** attribute is set to 0 is omitted.

(A)

Feature f	neutral$_f$	min$_f$	max$_f$	ϕ_f	δ_f^i
`one`	1	1	1	max	0
`width`	0	0	n	+	1
`surface`	0	$-\infty$	$+\infty$	+	x_i
`max`	$-\infty$	$-\infty$	$+\infty$	max	x_i
`min`	$+\infty$	$-\infty$	$+\infty$	min	x_i
`range`	0	0	$+\infty$		x_i

(B)

Aggregator g	default$_{g,f}$
`Max`	min$_f$
`Min`	max$_f$
`Sum`	0

initialisation	$R \leftarrow \text{default}_{g,f}$	$C \leftarrow \text{default}_{g,f}$	$D \leftarrow \text{neutral}_f$
return	$g(R, C)$		
semantic letters	**accumulator updates** *update of R*	*update of C*	*update of D*
out$_\text{r}$			$D \leftarrow \text{neutral}_f$
out$_\text{a}$	$R \leftarrow g(R, C)$	$C \leftarrow \text{default}_{g,f}$	$D \leftarrow \text{neutral}_f$
maybe$_\text{b}$			$D \leftarrow \phi_f\left(D, \delta_f^i\right)$
maybe$_\text{a}$			$D \leftarrow \phi_f\left(D, \delta_f^i\right)$
found		$C \leftarrow \phi_f\left(D, \delta_f^i\right)$	$D \leftarrow \text{neutral}_f$
found$_\text{e}$	$R \leftarrow g\left(R, \phi_f\left(D, \delta_f^i\right)\right)$		$D \leftarrow \text{neutral}_f$
in		$C \leftarrow \phi_f\left(C, \phi_f\left(D, \delta_f^i\right)\right)$	$D \leftarrow \text{neutral}_f$
out			

(C)

Fig. 6 (A) Features: their neutral, minimum, and maximum values, and the function $fv_i \leftarrow \phi_f(fv_{i-1}, \delta_f^i)$ used for computing the feature value of the feature f while reading variable x_i of its e-occurrence (δ_f^i provides the contribution of x_i with respect to feature f); (B) Aggregation operators with their corresponding default values; (C) Decoration table used for synthesising the feature automaton from its seed transducer and from an aggregation operator g and a feature f different from `range` (a slightly different function ϕ_f and different decoration table is used for `range`).

then e-occurrences and i-occurrences coincide. The **initialisation** row declares three accumulators R, C, and D, and their default values defined in Part B of Figure 6:

- R records the aggregated value of the features of the e-occurrences that were already completed, i.e., a **found**$_\text{e}$ or an **out**$_\text{a}$ was encountered.
- C stores the feature value of the current e-occurrence on which we did not yet reach the end, i.e., a **found** was encountered but we are still waiting for an **out**$_\text{a}$.
- D contains the feature value of the current potential part of an e-occurrence, i.e., the current semantic letter is a **maybe**$_\text{b}$ or a **maybe**$_\text{a}$.

The **return** row of Table C of Figure 6 provides the final result returned by the automaton (denoted as a green box in the generated graphs), the aggregation between the aggregated value R of the feature values of the e-occurrences already completed, and the feature value C of the current e-occurrences. Finally, the other rows of Table C give for each semantic letter of the output alphabet of a transducer the corresponding accumulator updates of R, C, and D:

- $\tau_i =$ **out**$_\text{r}$ indicates we are just after a **maybe**$_\text{b}$ and that index i is not part of an e-occurrence. We reinitialise the potential feature accumulator D.
- $\tau_i =$ **out**$_\text{a}$ means we discover the end of an e-occurrence, i.e., we are just after a **found**, an **in** or a **maybe**$_\text{a}$. Using the aggregation operator g we update the result

accumulator R in order to take into account the current feature value C, and we reinitialise the current feature and the potential feature values C and D.
- τ_i = **maybe**$_{\text{b}}$ (respectively **maybe**$_{\text{a}}$) means we are on a potential part of an e-occurrence. Using the ϕ_f function of Table A, we update the potential feature value D associated with the start (respectively the end) of the potential e-occurrence by taking into account x_i.
- τ_i = **found** means we just found an e-occurrence but do not know yet its end. We are copying the potential feature value D corresponding to the seen **maybe**$_{\text{b}}$ (taking also into account the current position x_i) to the current feature value C, and we reinitialise D.
- τ_i = **found**$_{\text{e}}$ means we just found an e-occurrence and we know that we reached its end. Using the aggregation operator g we update the result accumulator R with respect to the potential feature value D associated with the start of the potential e-occurrence corresponding to a sequence of **maybe**$_{\text{b}}$, and reinitialise D. Note that C does not need to be reinitialised since it was kept untouched while producing the **maybe**$_{\text{b}}$.
- τ_i = **in** means that the current e-occurrence is extended. We update the current feature value C taking into account both the potential feature accumulator D as well as x_i, and reinitialise D to the neutral element of f.
- τ_i = **out** means we are just after a **found**$_{\text{e}}$, an **out**, an **out**$_{\text{a}}$, or an **out**$_{\text{r}}$ that is not part of any e-occurrence. We do not generate any accumulator updates since **found**$_{\text{e}}$ and **out**$_{\text{a}}$ already took care of the update with respect to the end of an e-occurrence, and since **out**$_{\text{r}}$ took care of the update of D with respect to a **maybe**$_{\text{b}}$.

Example 6 Part A of Figure 7 provides the parameterised automaton obtained by applying Table C of Figure 6 to the transducer of the `peak` pattern defined in Part D of Figure 4, while Part B of Figure 7 instantiates the parameterised automaton to the automaton of the MIN_WIDTH_PEAK constraint.

Constraint checkers are synthesised by generating a big switch with respect to the states of the generated automata and the symbols of the input alphabet $\{<, =, >\}$, where each case of the switch corresponds to a transition of the automaton.

Proposition 1 (Computation correctness) *Given (1) a well-formed seed transducer $\mathcal{S}$, (2) a feature $f \in \{\texttt{one}, \texttt{width}, \texttt{surface}, \texttt{min}, \texttt{max}\}$, (3) an aggregator $g \in \{\texttt{Sum}, \texttt{Min}, \texttt{Max}\}$, (4) a time series $x_0, x_1, \ldots, x_{n-1}$ of integer constants and (5) the corresponding signature sequence $s = s_0, s_1, \ldots, s_{n-2}$, the decoration table given in Part (C) of Figure 6 computes the aggregation with respect to g of feature f applied on the t-occurrences of $\mathcal{S}$ for s.*

Proof The proof consists of three elements: Two nested inductions over the input word and a case analysis for the concrete features and aggregators. The inner induction, based on the t-occurrence of Definition 4, checks that feature computations are initialised correctly and that the feature value inside a single occurrence of a pattern is computed correctly (see Lemmas 1,2,3), the outer induction checks that the aggregation of multiple occurrences computes the aggregated value correctly (see Lemma 4). The third part demonstrates that the abstract feature operations compute the correct values for the concrete features (see Lemma 5). □

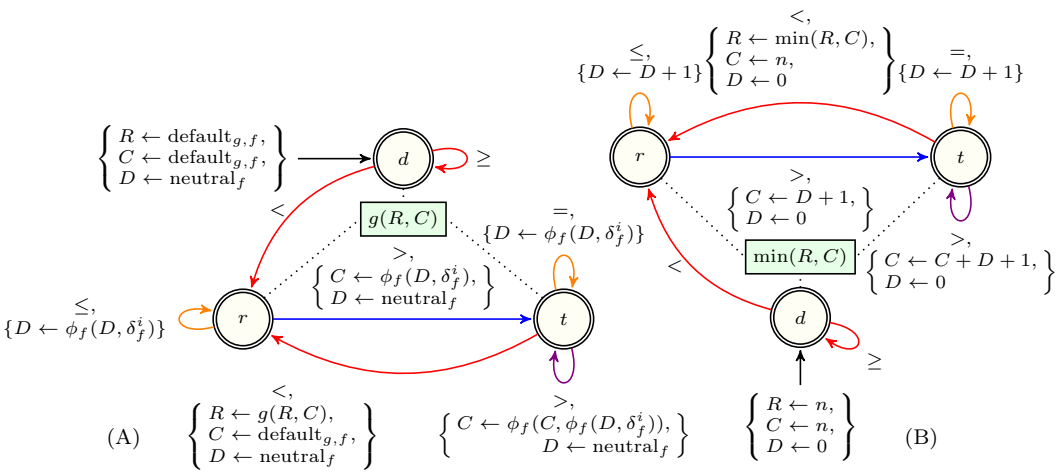


Fig. 7 (A) Aggregation of pattern features automaton of the `peak` pattern with respect to the aggregation operator g applied to feature f, (B) corresponding concrete automaton for $g = \min$ and $f =$ `width` for the MIN_WIDTH_PEAK($result, \langle x_0, x_1, \ldots, x_{n-1}\rangle$) constraint obtained by using Tables A and B of Figure 6.

Notation 1 *To simplify the reading we use in the following Lemmas the n-ary notation for function ϕ_f, e.g. $\phi_f(\phi_f(a, b), c)$ is denoted by $\phi_f(a, b, c)$. The empty word is denoted by ϵ.*

Lemma 1 (state of counter D just before entering a t-occurrence) *Just before entering a t-occurrence, the counter D is set to* neutral_f.

Proof According to Definition 4, a t-occurrence corresponds to the indices of the regular expressions '$\mathbf{maybe}_{\text{b}}^*\mathbf{found}_{\text{e}}$' and '$\mathbf{maybe}_{\text{b}}^*\mathbf{found}(\mathbf{maybe}_{\text{a}}^*\mathbf{in}^+)^*$' used to compute the corresponding feature value. From the wellformedness hypothesis of the seed transducer $\mathcal{S}$, a t-occurrence is (1) either located at the very start of a time series, (2) or immediately preceded by one of the following semantic letters: $\mathbf{out}$, $\mathbf{out}_{\text{r}}$, $\mathbf{out}_{\text{a}}$, $\mathbf{found}_{\text{e}}$.

- In case (1) the initialisation statement of the decoration table given in part (C) of Fig. 6 sets counter D to the value indicated by Lemma 1.
- – In case (2) assume that the immediately preceding semantic letter is $\mathbf{out}_{\text{r}}$, $\mathbf{out}_{\text{a}}$ or $\mathbf{found}_{\text{e}}$. Then the initialisation statement of the decoration table resets counter D to the value indicated by Lemma 1.
  - In case (2) assume that the immediately preceding semantic letter is $\mathbf{out}$. Then from the wellformedness of $\mathcal{S}$ we have that the preceding semantic letter will be itself preceded by a word of the form '$(\mathbf{out}_{\text{r}}|\mathbf{out}_{\text{a}}|\mathbf{found}_{\text{e}}|\epsilon)\mathbf{out}^*$'. The part $(\mathbf{out}_{\text{r}}|\mathbf{out}_{\text{a}}|\mathbf{found}_{\text{e}}|\epsilon)$ resets counter D to the value indicated by Lemma 1 and the $\mathbf{out}^*$ leaves counter D unchanged. □

Lemma 2 (feature value for a t-occurrence of the form '$\mathbf{maybe}_{\text{b}}^*\mathbf{found}_{\text{e}}$') *Given a t-occurrence of the form '$\mathbf{maybe}_{\text{b}}^*\mathbf{found}_{\text{e}}$' starting at position i and ending at position j, the computed feature value is $\phi_f(\text{neutral}_f, \delta_f^i, \delta_f^{i+1}, \ldots, \delta_f^j)$.*

Proof
• First, just before entering a t-occurrence, by Lemma 1, counter D is assigned value neutral_f.
• Second, from the entry of the decoration table attached to $\mathbf{maybe}_\text{b}$, we have that just after the last $\mathbf{maybe}_\text{b}$ of the t-occurrence, counter D will be assigned value $\phi_f(\text{neutral}_f, \delta_f^i, \delta_f^{i+1}, \ldots, \delta_f^{j-1})$.
• Finally, from the entry of the decoration table attached to $\mathbf{found}_\text{e}$, the feature value is $\phi_f(D, \delta_f^j)$ (i.e. the second argument of the aggregation operator g), which taking into account the value of D leads to the expected value $\phi_f(\text{neutral}_f, \delta_f^i, \delta_f^{i+1}, \ldots, \delta_f^j)$.
□

Lemma 3 (feature value for a t-occurrence '$\mathbf{maybe}_\text{b}^*\mathbf{found}$ $(\mathbf{maybe}_\text{a}^*\mathbf{in}^+)^*$') *Given a t-occurrence of the form '$\mathbf{maybe}_\text{b}^*\mathbf{found}(\mathbf{maybe}_\text{a}^*\mathbf{in}^+)^*$' starting at position i and ending at position j, the computed feature value is $\phi_f(\text{neutral}_f, \delta_f^i, \delta_f^{i+1}, \ldots, \delta_f^j)$.*

Proof
• First, by Lemma 1, the counter D is assigned value neutral_f just before entering a t-occurrence.
• Second, from the entry of the decoration table attached to $\mathbf{maybe}_\text{b}$, we have that just after the last $\mathbf{maybe}_\text{b}$ of the t-occurrence the counter D will be assigned value $\phi_f(\text{neutral}_f, \delta_f^i, \delta_f^{i+1}, \ldots, \delta_f^{k-1})$, where k is the position of the last $\mathbf{maybe}_\text{b}$.
• Third, from the entry of the decoration table attached to **found**, we have that counter C will be assigned value $\phi_f(\text{neutral}_f, \delta_f^i, \delta_f^{i+1}, \ldots, \delta_f^k)$ and D will be reset to value neutral_f.
• Fourth, assume we are at stage '$\mathbf{maybe}_\text{b}^*\mathbf{found}(\mathbf{maybe}_\text{a}^*\mathbf{in}^+)^*\mathbf{maybe}_\text{a}^*$'. We have that counter C will be assigned value $\phi_f(\text{neutral}_f, \delta_f^i, \delta_f^{i+1}, \ldots, \delta_f^{k_1})$ and D will be assigned value $\phi_f(\text{neutral}_f, \delta_f^{k_1+1}, \delta_f^{k_1+2}, \ldots, \delta_f^{k_2})$, where k_1 and k_2 are respectively the length of the words '$\mathbf{maybe}_\text{b}^*\mathbf{found}(\mathbf{maybe}_\text{a}^*\mathbf{in}^+)^*$' and '$\mathbf{maybe}_\text{a}^*$'. This stems from the entry of the decoration table attached to $\mathbf{maybe}_\text{a}$ and from the fact that counter D is reset to neutral_f both after a **found** and after an **in**.
• Fifth, assume we are at stage '$\mathbf{maybe}_\text{b}^*\mathbf{found}(\mathbf{maybe}_\text{a}^*\mathbf{in}^+)^*\mathbf{maybe}_\text{a}^*\mathbf{in}$'. Then counter C will be assigned value $\phi_f(\text{neutral}_f, \delta_f^i, \delta_f^{i+1}, \ldots, \delta_f^\ell)$ and D will be reset to neutral_f, where ℓ is the length of '$\mathbf{maybe}_\text{b}^*\mathbf{found}(\mathbf{maybe}_\text{a}^*\mathbf{in}^+)^*\mathbf{maybe}_\text{a}^*\mathbf{in}$'. This stems from the entry of the decoration table attached to **in**, which transfers the content of D (corresponding to a stretch of $\mathbf{maybe}_\text{a}$) to C. □

Lemma 4 (aggregation of feature values) *The aggregation of the feature values is sound.*

Proof Depending on whether we have a t-occurrence of the form '$\mathbf{maybe}_\text{b}^*\mathbf{found}_\text{e}$' or of the form '$\mathbf{maybe}_\text{b}^*\mathbf{found}$ $(\mathbf{maybe}_\text{a}^*\mathbf{in}^+)^*$' the aggregation takes place respectively at:

1. A $\mathbf{found}_\text{e}$ where counter R is updated wrt the current feature value $\phi_f(D, \delta_f^j)$, where j is the last position of the current t-occurrence.
2. An $\mathbf{out}_\text{a}$ (see the automaton defining the wellformedness given in Part (A) of Figure 4) or when exiting the automaton (i.e. see the return statement of the decoration table). In both cases the counter R is updated wrt the current feature value that is recorded in C (see Lemma 3).

Finally, note that the counter C contains its default value $\text{default}_{g,f}$ when we encounter a $\textbf{found}_{\text{e}}$ and consequently does not need to be reset to $\text{default}_{g,f}$ when we are on a $\textbf{found}_{\text{e}}$. This is because counter C is only used when we have a **found** (and on an **in** after a **found**) and is reset to its default value while exiting the corresponding pattern on an $\textbf{out}_{\text{a}}$. □

Lemma 5 (soundness of the computation of concrete features) *The computation of concrete features is sound.*

Proof Each row of Table (A) of Figure 6 defines the computation of one given concrete feature. Given a t-occurrence that is defined from position i to position j the computation of a concrete feature value starts by initialising counter D to the neutral element neutral_f of feature f as defined by the second column of Table (A). On the first position i of the t-occurrence we get the feature value $\phi_f(\text{neutral}_f, \delta_f^i)$, respectively equal to 1, 1, x_i, x_i, x_i, which are the expected results for a t-occurrence involving only one single position depending on whether f is equal to `one`, `width`, `surface`, `max` or `min`. On the last position j of the t-occurrence we get the feature value 1, $j - i + 1$, $\sum_{k\in[i,j]} x_k$, $\max_{k\in[i,j]} x_k$, $\min_{k\in[i,j]} x_k$, which are the expected feature values. □

A relevant question is how efficient our synthesised automata and checkers are compared to the manually written code already in the catalogue. To answer that question, we selected a few constraints, and for each one, we perform 100 runs for the manually defined and for the synthesised versions by

- generating a sequence of 1,000 domain variables with random domains, posting the automaton with a fixed aggregated feature value, searching for a solution with a 10s time limit, repeating until a solution is found,
- generating a sequence of 10,000,000 integers and letting the checker compute the aggregated value,

In Table 2, μ_{gA} and σ_{gA} denote the geometric mean and standard deviation of $\frac{\text{synthesised}}{\text{manual}}$ automata run time, and μ_{gC} and σ_{gC} similarly for checker run time. The last column indicates the bounds of the input sequence. The experiments were run in SICStus Prolog 4.3.1 on a quad core 2.8 GHz Intel Core i7-860 machine with 8MB cache per core, running Ubuntu Linux.

Table 2 Run-Time Performance Evaluation for Selected Constraints

constraint	μ_{gA}	σ_{gA}	μ_{gC}	σ_{gC}	**domain**
NB_PEAK	1.14	0.18	2.66	0.01	1..3
NB_INFLEXION	1.19	0.05	2.73	0.02	1..3
MIN_SURF_PEAK	1.09	0.06	1.10	0.03	1..10
MAX_RANGE_INCREASING_SEQUENCE	0.30	0.06	1.54	0.02	1..10

For the NB_PEAK constraint, the generated automaton constraint propagator is 14% slower than the manually generated code, while the checker is 2.66 times slower. The results indicate that the overhead of generated code compared to manually defined propagators is quite limited, while the overhead of the generated checkers is higher, but not prohibitive. This overhead can be further reduced by applying code simplification

techniques on the generated code. In particular, for certain combinations of pattern, feature and aggregator we can collapse certain nodes in the automaton, eliminate some counters completely, and simplify the updates of the remaining counters.

6 Examples of Use

A natural use of the time-series constraints is descriptive, using them to generate features for some further analysis, for example clustering [20] or similarity analysis [16, 22]. But they can also be used actively, to generate new time series from existing ones. We now discuss a few examples.

6.1 Generation of Similar Curves

Our previous work on time series [8] was motivated by an application problem from EDF, the largest French electricity supplier. We used the daily power output curves of each power station to generate a model of the capability of the generator for inclusion in the Unit Commitment Model, which optimises the overall generation cost. At the moment, the manual definition and maintenance of these generator specific models is both resource consuming and a potential source of errors. For the earlier work, we had manually generated 30 time series constraints to capture important properties of the series. As part of the analysis of the results obtained, the end-users described several additional constraints that should be considered. For example, for some generator types, when the power output reaches a peak, that level should be maintained at least for a certain number of time periods. Instead of adding more constraints by hand, we decided on a more systematic reconstruction of potential time-series constraints. For the constraint mentioned, the MIN_WIDTH_PLATEAU constraint captures the required behaviour. In the EDF use case, we use the constraint checkers to detect properties that are similar during most days of a time period. We then can use the generated constraint model to generate similar time series that satisfy other properties, like total cost or required overall capacity.

In a different application, we use time-series constraints to capture the capacity of irregularly run clinics in a hospital. Many clinics are run with a regular pattern, like Tuesday of every second week, but others are run on a more ad-hoc basis. When simulating the hospital capacity, we can use the extracted constraint model to repeatedly generate typical, but different, run patterns for these clinics, without knowing the exact future timing of the events.

6.2 Data Correction: Identifying and Fixing Problems in Data

In the context of a European project about energy management for buildings [3], we use our time-series constraints to detect and correct errors in the input data collected from data sensors. For some wireless temperature sensors in one of the buildings for example, typical failure modes are either a stuck value, where the same temperature value is returned in multiple consecutive time periods, or a sudden drop in value from

[3] http://www.campus21-project.eu/index.php/en/

one time period to the next. We use the footprint constraints to identify the terrace or gorge pattern, and the overall constraint model to repair the time series with plausible values, only introducing variables for time periods where a problem was detected. The use of constraint models to describe and (partially) repair faulty sensor data is a more declarative and flexible solution than the ad-hoc code that is used in most existing building management systems, while being simpler than other methods to estimate missing values in time series [14].

7 Conclusion

By using the concept of seed transducer, we have shown how to synthesise systemically a variety of time-series constraints. From 20 patterns and their seed transducers, which were created manually, we have synthesised over 600 constraints, which combine systematically pattern, feature and aggregation operators. This is achieved by providing for each class of constraints, two of which are shown in this paper, a decoration table that maps the semantic letters of the output alphabet of the seed transducer to accumulator updates of the synthesised automaton-based constraints. This work contributes, in the context of time-series constraints, to the systematic reconstruction of the Global Constraint Catalogue that we have previously advocated [7]. These constraints are not only useful to describe properties of time series, but can be used in a wide range of applications to generate (parts of) new time series based on previously observed samples.

Acknowledgements

The first author was partially supported by the Gaspard-Monge program that initially motivated this research. The first and third authors were partially supported by the European H2020 FETPROACT-2014 project "GRACeFUL". The last author was partially supported by a senior researcher chair by the Région Pays de la Loire and by the EU FET grant ICON (project number 284715). We thank Pierre Flener for very actively helping clarifying the different notions of pattern occurrences introduced in the paper and the reviewers for their suggestions.

References

1. Abney, S.: Partial parsing via finite-state cascades. Natural Language Engineering 2(4), 337–344 (1996)
2. Beldiceanu, N., Carlsson, M., Debruyne, R., Petit, T.: Reformulation of global constraints based on constraints checkers. Constraints 10(4), 339–362 (2005)
3. Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T.: Global constraint catalogue: Past, present and future. Constraints 12(1), 21–62 (2007)
4. Beldiceanu, N., Carlsson, M., Flener, P., Pearson, J.: On the reification of global constraints. Constraints 18(1), 1–6 (2013)
5. Beldiceanu, N., Carlsson, M., Flener, P., Rodríguez, M.A.F., Pearson, J.: Linking prefixes and suffixes for constraints encoded using automata with accumulators. In: O'Sullivan, B. (ed.) Principles and Practice of Constraint Programming (CP 2014). LNCS, vol. 8656, pp. 142–157. Springer (2014)
6. Beldiceanu, N., Carlsson, M., Rampon, J.X.: Global constraint catalog, 2nd edition (revision a). Tech. Rep. T2012-03, Swedish Institute of Computer Science (2012), current version available at http://sofdem.github.io/gccat/

7. Beldiceanu, N., Flener, P., Monette, J.N., Pearson, J., Simonis, H.: Toward sustainable development in constraint programming. Constraints 19(2), 139–149 (2014)
8. Beldiceanu, N., Ifrim, G., Lenoir, A., Simonis, H.: Describing and generating solutions for the EDF unit commitment problem with the ModelSeeker. In: Schulte, C. (ed.) Principles and Practice of Constraint Programming (CP 2013). LNCS, vol. 8124, pp. 733–748. Springer (2013)
9. Beldiceanu, N., Simonis, H.: A constraint seeker: Finding and ranking global constraints from examples. In: Lee, J. (ed.) Principles and Practice of Constraint Programming (CP 2011). LNCS, vol. 6876, pp. 12–26. Springer (2011)
10. Beldiceanu, N., Simonis, H.: A model seeker: Extracting global constraint models from positive examples. In: Milano, M. (ed.) Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7514, pp. 141–157. Springer (2012), http://dx.doi.org/10.1007/978-3-642-33558-7_13
11. Berstel, J.: Transductions and Context-Free Languages. Teubner (1979)
12. Carlsson, M., et al.: SICStus Prolog User's Manual. Swedish Institute of Computer Science, 4.3.1 edn. (November 2014), current version available at https://sicstus.sics.se/sicstus/docs/latest4/pdf/sicstus.pdf
13. Fu, T.: A review on time series data mining. Engineering Applications of Artificial Intelligence 24(1), 164 – 181 (2011), http://www.sciencedirect.com/science/article/pii/S0952197610001727
14. Fung, D.S.C.: Methods for the Estimation of Missing Values in Time Series. Master's thesis, Edith Cowan University, Perth, Australia (2006)
15. Gent, I.P., Jefferson, C., Linton, S., Miguel, I., Nightingale, P.: Generating custom propagators for arbitrary constraints. Artificial Intelligence 211, 1–33 (2014)
16. Goldin, D.Q., Kanellakis, P.C.: On similarity queries for time-series data: Constraint specification and implementation. In: Montanari, U., Rossi, F. (eds.) Principles and Practice of Constraint Programming (CP 1995). LNCS, vol. 976, pp. 137–153. Springer (1995)
17. Guns, T., Nijssen, S., De Raedt, L.: Itemset mining: A constraint programming perspective. Artificial Intelligence 175(12–13), 1951–1983 (2011)
18. Harvey, A.: Forecasting, Structural Time Series Models and the Kalman Filter. Cambridge University Press (February 1991)
19. Laurière, J.L.: Constraint propagation or automatic programming. Tech. Rep. 19, IBP-Laforia (1996), in French, available at https://www.lri.fr/~sebag/Slides/Lauriere/Rabbit.pdf
20. Liao, T.W.: Clustering of time series data - a survey. Pattern Recognition 38(11), 1857–1874 (2005), http://dx.doi.org/10.1016/j.patcog.2005.01.025
21. Nhon, D.T., Wilkinson, L.: TimeExplorer: Similarity search time series by their signatures. In: Bebis, G., Boyle, R., Parvin, B., Koracin, D., Li, B., Porikli, F., Zordan, V.B., Klosowski, J.T., Coquillart, S., Luo, X., Chen, M., Gotz, D. (eds.) 9th International Symposium on Advances in Visual Computing (ISVC 2013). LNCS, vol. 8033, pp. 280–289. Springer (2013)
22. Perng, C.S., Wang, H., Zhang, S.R., Parker, D.S.: Landmarks: A new model for similarity-based pattern querying in time series databases. In: 16th International Conference on Data Engineering (ICDE 2000). pp. 33–42. IEEE (2000)
23. Ratanamahatana, C., Lin, J., Gunopulos, D., Keogh, E., Vlachos, M., Das, G.: Mining time series data. In: Maimon, O., Rokach, L. (eds.) Data Mining and Knowledge Discovery Handbook, pp. 1049–1077. Springer US (2010), http://dx.doi.org/10.1007/978-0-387-09823-4_56
24. Sakarovitch, J.: Elements of Language Theory. Cambridge University Press (2009)
25. Smith, D.R., Westfold, S.J.: Toward the synthesis of constraint solvers. Tech. Rep. TR-1311, Kestrel Institute (2013), available at http://www.kestrel.edu/home/people/smith/pub/CW-report.pdf
26. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjørner, N.: Symbolic finite state transducers: algorithms and applications. In: Field, J., Hicks, M. (eds.) Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012. pp. 137–150. ACM (2012)